**DESIGN PATTERS AND PRINCIPLES**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

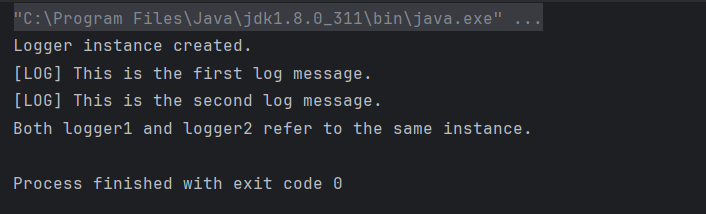
**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

CODE:

class Logger {  
  
 // Step 1: Create a class named Logger that has a private static instance of itself  
 private static Logger *instance*;  
  
 // Step 2: Ensure the constructor of Logger is private  
 private Logger() {  
 System.*out*.println("Logger instance created.");  
 }  
  
 // Step 3: Provide a public static method to get the instance of the Logger class  
 public static Logger getInstance() {  
 if (*instance* == null) {  
 *instance* = new Logger(); // Lazy initialization  
 }  
 return *instance*;  
 }  
  
 // A sample method for logging  
 public void log(String message) {  
 System.*out*.println("[LOG] " + message);  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
  
 // Get Logger instance  
 Logger logger1 = Logger.*getInstance*();  
 logger1.log("This is the first log message.");  
  
 // Get Logger instance again  
 Logger logger2 = Logger.*getInstance*();  
 logger2.log("This is the second log message.");  
  
 // Verify if both references point to the same object  
 if (logger1 == logger2) {  
 System.*out*.println("Both logger1 and logger2 refer to the same instance.");  
 } else {  
 System.*out*.println("Different Logger instances were created.");  
 }  
 }  
}

OUTPUT:



**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

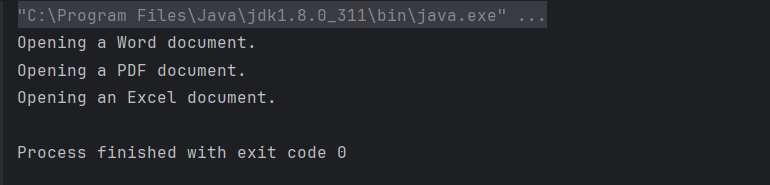
**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

CODE:

interface Document {  
 void open();  
}  
  
// File: WordDocument.java  
class WordDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening a Word document.");  
 }  
}  
  
// File: PdfDocument.java  
class PdfDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening a PDF document.");  
 }  
}  
  
// File: ExcelDocument.java  
class ExcelDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening an Excel document.");  
 }  
}  
  
// File: DocumentFactory.java  
abstract class DocumentFactory {  
 public abstract Document createDocument();  
}  
  
// File: WordDocumentFactory.java  
class WordDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}  
  
// File: PdfDocumentFactory.java  
class PdfDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}  
  
// File: ExcelDocumentFactory.java  
class ExcelDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new ExcelDocument();  
 }  
}  
  
// File: Main.java  
public class Main {  
 public static void main(String[] args) {  
  
 DocumentFactory wordFactory = new WordDocumentFactory();  
 Document wordDoc = wordFactory.createDocument();  
 wordDoc.open(); // Output: Opening a Word document.  
  
 DocumentFactory pdfFactory = new PdfDocumentFactory();  
 Document pdfDoc = pdfFactory.createDocument();  
 pdfDoc.open(); // Output: Opening a PDF document.  
  
 DocumentFactory excelFactory = new ExcelDocumentFactory();  
 Document excelDoc = excelFactory.createDocument();  
 excelDoc.open(); // Output: Opening an Excel document.  
 }  
}

OUTPUT:



**ALGORITHMS\_ DATA STRUCTURES**

**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

CODE:

STEP 1: **What is Big O Notation?**

**Big O notation** describes how an algorithm’s runtime or space requirements grow relative to the input size (**n**). It helps estimate the **efficiency** of an algorithm.

| **Complexity** | **Description** |
| --- | --- |
| O(1) | Constant time |
| O(log n) | Logarithmic time |
| O(n) | Linear time |
| O(n log n) | Linearithmic time |
| O(n²) | Quadratic time |

**Search Scenarios:**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

Binary search **requires sorted data**, but performs significantly better on large datasets.

STEP 2: Setup

// File: Product.java

public class Product {

int productId;

String productName;

String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String toString() {

return "[" + productId + ", " + productName + ", " + category + "]";

}

}

Step 3: Implementation

// File: SearchUtility.java

import java.util.Arrays;

import java.util.Comparator;

public class SearchUtility {

// Linear Search by Product Name

public static Product linearSearch(Product[] products, String name) {

for (Product product : products) {

if (product.productName.equalsIgnoreCase(name)) {

return product;

}

}

return null;

}

// Binary Search by Product Name (requires sorted array)

public static Product binarySearch(Product[] products, String name) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int comparison = products[mid].productName.compareToIgnoreCase(name);

if (comparison == 0)

return products[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

// Sort products by productName

public static void sortByName(Product[] products) {

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

}

}

Step 4: Test and Analyze

// File: Main.java

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Sneakers", "Footwear"),

new Product(3, "Phone", "Electronics"),

new Product(4, "Backpack", "Accessories"),

new Product(5, "Watch", "Accessories")

};

System.out.println("🔍 Linear Search for 'Phone':");

Product foundLinear = SearchUtility.linearSearch(products, "Phone");

System.out.println(foundLinear != null ? foundLinear : "Product not found");

System.out.println("\n🔍 Binary Search for 'Phone':");

SearchUtility.sortByName(products); // Sort before binary search

Product foundBinary = SearchUtility.binarySearch(products, "Phone");

System.out.println(foundBinary != null ? foundBinary : "Product not found");

}

}

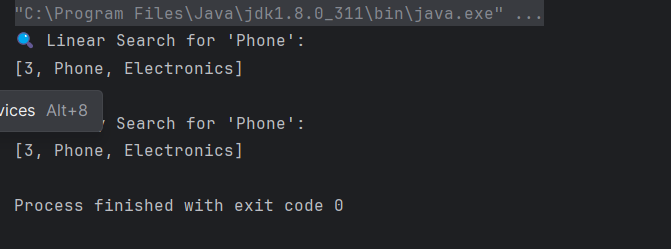
**Analysis: Linear vs Binary Search**

| **Metric** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Time Complexity | O(n) | O(log n) |
| Requires Sorting | No | Yes (O(n log n)) |
| Performance on Large | Slower | Much faster |
| Simplicity | Simple | Needs sorted input |

**Which is better for your platform?**

* Using **Linear Search** for **small data sets** or **unsorted data**.
* Using **Binary Search** for **larger data sets** that can be sorted once and searched multiple times.

**Binary Search is preferred** for performance optimization in an e-commerce search engine where speed matters.



**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

CODE:

Step 1: **What is Recursion?**

**Recursion** is a programming technique where a function **calls itself** to solve a problem.  
It works well for problems that can be broken down into **smaller, similar subproblems**, such as:

* Factorial
* Fibonacci series
* Tree traversals
* Financial growth over time

**How It Helps**

In financial forecasting, recursion can model **compound growth**, where each year builds on the previous one.

Step 2: Setup – Future Value Calculation Method

future value FV using the formula:

FV=PV×(1+r)nFV = PV \times (1 + r)^nFV=PV×(1+r)n

Where:

* **PV** = Present Value
* **r** = Growth rate (as a decimal)
* **n** = Number of years

Step 3: Recursive Implementation

// File: FinancialForecast.java

public class FinancialForecast {

// Recursive method to calculate future value

public static double calculateFutureValue(double presentValue, double rate, int years) {

// Base case

if (years == 0) {

return presentValue;

}

// Recursive case

return calculateFutureValue(presentValue \* (1 + rate), rate, years - 1);

}

public static void main(String[] args) {

double presentValue = 1000.0; // initial investment

double rate = 0.05; // 5% annual growth

int years = 5;

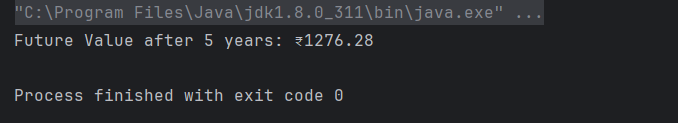
double futureValue = calculateFutureValue(presentValue, rate, years);

System.out.printf("Future Value after %d years: ₹%.2f%n", years, futureValue);

}

}

Output:



Step 4: Analysis

**Time Complexity:**

* The function is called **once per year**, so the **time complexity is O(n)** where n is the number of years.

**Optimization Tip:**

* While this recursive solution is fine for small n, for large values it could lead to **stack overflow**.
* You can convert it to **tail recursion** or use an **iterative solution** for better efficiency and memory use.

Optimized version:

public static double calculateFutureValueIterative(double presentValue, double rate, int years) {

for (int i = 0; i < years; i++) {

presentValue \*= (1 + rate);

}

return presentValue;

}